42390P10807 PATENT

UNITED STATES PATENT APPLICATION

for

TRACKING OPERATING SYSTEM PROCESS AND THREAD EXECUTION AND VIRTUAL MACHINE EXECUTION IN HARDWARE OR IN A VIRTUAL MACHINE MONITOR

Applicants:

Erik Cota-Robles
Sebastian Schoenberg
Stalinselvaraj Jeyasingh
Alain Kagi
Michael Kozuch
Gilbert Neiger
Richard Uhlig

prepared by:

BLAKELY, SOKOLOFF, TAYLOR & ZAFMAN
12400 Wilshire Boulevard
Los Angeles, CA 90026-1026
(408) 720-8598

EXPRESS MAIL CERTIFICATE OF MAILING

"Express Mail" mailing label number EL431888639US
Date of Deposit August 15, 2001
I hereby certify that this paper or fee is being deposited with the United States Postal Service "Express Mail Post Office to Addressee" service under 37 CFR 1.10 on the date indicated above and is addressed to the Commissioner of Patents and Trademarks, Washington, D.C. 20231.

Michelle Begay
(Typed or printed name of person mailing paper or fee)

Michelle Begay
(Signature of person mailing paper or fee)

TRACKING OPERATING SYSTEM PROCESS AND THREAD EXECUTION AND VIRTUAL MACHINE EXECUTION IN HARDWARE OR IN A VIRTUAL MACHINE MONITOR

RELATED APPLICATIONS

This application is related to U.S. Patent application serial number 09/541,444 titled REAL-TIME SCHEDULING OF VIRTUAL MACHINES, filed on March 31, 2000, and U.S. Patent application serial number 09/752,134 titled NEW PROCESSOR MODE FOR LIMITING THE OPERATION OF GUEST SOFTWARE RUNNING ON A VIRTUAL MACHINE SUPPORTED BY A VIRTUAL MACHINE MONITOR, filed on December 27, 2000, both of which are assigned to the assignee of the present application.

FIELD OF THE INVENTION

This invention relates generally to virtual machine environments, and more particularly to scheduling virtual machines within those environments.

BACKGROUND OF THE INVENTION

An Operating System (OS) is a software program that controls physical computer hardware (e.g., a processor, memory, and disk and CD-ROM drives) and presents application programs with a unified set of abstract services (e.g., a file system). Modern OSs typically multi-task among several application programs, each of which executes in a separate process, and many enable application programs to multi-task among several "threads" that share the same process address space.

Modern processors frequently have "performance counters," software-configurable registers that count occurrences of various performance "events." Typical events include

"instructions retired" and "processor cycles," the ratio of which is the well-known metric Instructions Per Clock (IPC), as well as various types of cache misses. Performance monitoring applications use these counters and events to evaluate and tune the performance of other applications. In a multi-tasking environment, performance monitoring applications must distinguish events such as cache misses that occur in one program or thread from those that occur in other programs or threads. Since hardware counters count these events, the inability to track OS process and thread execution in hardware limits the usefulness of the performance monitoring applications.

A Virtual Machine Monitor (VMM) is a software program that controls physical computer hardware (e.g., a processor, memory, and disk and CD-ROM drives) and presents programs executing within a Virtual Machine (VM) with the illusion that they are executing on real physical computer hardware (a "bare" machine, e.g., a processor, memory and a disk drive). Each VM typically functions as a self-contained platform, controlled by a "guest" OS (i.e., an OS hosted by the VMM), which executes as if it were running on a "bare" machine instead of within a virtual machine. Recursive VMs are VMs that are controlled by a VMM that is itself executing within a VM.

In a "real-time" application, computations upon data that is available at one substantially predetermined time should be completed by another substantially predetermined time. If an OS (or VMM) schedules a thread or process (or VM) with sufficient frequency and for sufficient duration that the thread or process (or VM) is able to complete its computations before their respective deadlines, the thread or process (or VM) is said to have received adequate scheduling Quality of Service (QoS). OSs and VMMs should schedule the computing resources of their real or virtual machine in such a fashion as to ensure that real-time applications receive adequate scheduling QoS.

A VMM can monitor scheduling QoS at the level of all applications within a VM as disclosed in the related application serial number 09/541,444. However, such monitoring cannot distinguish between real-time and non-real-time applications in the same VM, nor can it distinguish among recursive VMs in the same VM, leading to problems in providing adequate scheduling QoS. Furthermore, a system wide performance monitoring facility that executes as part of a VMM will need to distinguish events in one VM from those in another VM.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1A is a diagram illustrating one embodiment of a bare hardware environment;

Figure 1B is a diagram illustrating one embodiment of a virtual machine environment;

Figure 2A is a flowchart of one embodiment of a method of tracking state changes in the bare machine environment of Figure 1A;

Figures 2B-C are flowcharts of one embodiment of a method of tracking state changes in the virtual machine environment of Figure 1B;

Figure 3 is a block diagram of one embodiment of microcode components used in conjunction with the method of Figure 2A;

Figures 4A-B are diagrams of process/thread data structures used by the methods of Figures 2B-C;

Figure 5 is a block diagram illustrating one embodiment of a virtual machine monitor operating with the virtual machine environment of Figure 1A;

Figure 6 is a flowchart of one embodiment of a method of scheduling virtual

machines using the virtual machine monitor of Figure 5;

Figure 7 is a diagram illustrating one embodiment of a virtual machine environment that supports recursive virtualization;

Figures 8A-B are flowcharts of one embodiment of a method of tracking virtual machine changes in the virtual machine environment of Figure 7;

Figure 9A-B are diagrams of virtual machine data structures used by the method of Figures 8A-B;

Figure 10 is a flowchart of one embodiment of a method of scheduling recursive virtual machines within the environment of Figure 7; and

Figure 11 is a block diagram of a hardware computing platform suitable for hosting the machine environments of Figures 1A, 1B and 7.

DETAILED DESCRIPTION OF THE INVENTION

In the following detailed description of embodiments of the invention, reference is made to the accompanying drawings in which like references indicate similar elements, and in which is shown by way of illustration specific embodiments in which the invention may be practiced. These embodiments are described in sufficient detail to enable those skilled in the art to practice the invention, and it is to be understood that other embodiments may be utilized and that logical, mechanical, electrical, functional and other changes may be made without departing from the scope of the present invention. The following detailed description is, therefore, not to be taken in a limiting sense, and the scope of the present invention is defined only by the appended claims.

Furthermore, particular embodiments of methods of the invention are described in terms of executable instructions with reference to a series of flowcharts. Describing the

methods by reference to a flowchart enables one skilled in the art to develop such instructions to carry out the methods within suitably configured processing units. The executable instructions may be written in a computer programming language or may be embodied in firmware logic. The present invention is not described with reference to any particular programming language and it will be appreciated that a variety of programming languages may be used to implement the teachings of the invention as described herein. In addition, it is common in the art to speak of executable instructions as taking an action or causing a result. Such expressions are merely a shorthand way of saying that execution of the instructions by a computer causes the processor of the computer to perform an action or a produce a result.

The present invention infers software actions that result in transitions among operating system processes and threads as well as among virtual machines. These inferences can be made either by real hardware (i.e., a computer processor) or by a virtual machine monitor, in which case the virtual machines whose transitions are being inferred are recursive virtual machines whose transitions are controlled by a child virtual machine monitor that executes within one of the virtual machines controlled by the virtual machine monitor itself. In one of the embodiments described herein, hardware performance counters incorporate the techniques of this invention to distinguish the occurrence of performance "events" such as "instructions retired", "processor cycles" or "cache misses" in one operating system process or thread or in an entire virtual machine from those, respectively, in another operating system process or thread or in an entire virtual machine. In another embodiment, the invention is incorporated into a virtual machine monitor to enable a virtual machine system to guarantee adequate scheduling Quality of Service (QoS) to real-time applications executing in virtual machines that contain both real-time

and non-real-time applications. The use of the invention in a recursive virtualization system where a child virtual machine monitor controls multiple virtual machines, with one or more of the recursive virtual machines executing one or more real-time applications, and one or more recursive virtual machines executing one or more non-real-time applications is also described. The invention is not so limited by these embodiments, however, and the scheduling information also can be used, for example, by multiprocessor systems and multithreaded processors to better assign instruction streams to particular processors and/or hardware contexts.

Figure 1A illustrates a bare hardware system 150 within which various embodiments of the invention may be practiced. The bare hardware system 150 comprises a bare machine 101 that runs a computer operating system (OS) 157 as privileged software (e.g., in ring 0). Operating systems and their typical features and functionality are well-known by those skilled in the art. Bare machine 101 is a hardware computing platform that includes, at a minimum, a processing unit, a memory, and a bus coupling the processor and the memory. One embodiment of a hardware computing platform suitable for practicing the invention is illustrated in Figure 11 and described further below.

The OS 157 schedules processes A 153 and B 155 for execution on the bare machine 101. In addition OS 157 may directly schedule threads A-1 163, A-2 164, A-3 165 and B-2 166, for execution or it may allow the processes 153, 155 to schedule the threads 163, 164, 165, 166 themselves. The actual mechanism(s) used to schedule the processes and threads for execution on the bare machine are well-known by those skilled in the art.

Figure 1B illustrates a virtual machine system 100 within which various embodiments of the invention may be practiced. The virtual machine system 100

comprises a bare machine 101 hosting a virtual machine monitor (VMM) 107 that runs as privileged software (i.e., in ring 0). VMMs and their typical features and functionality are well-known by those skilled in the art and may be implemented, for example, in software, firmware, hardware or through a combination of them. Bare machine 101 is a hardware computing platform that includes, at a minimum, a processing unit, a memory, and a bus coupling the processor and the memory. One embodiment of a hardware computing platform suitable for practicing the invention is illustrated in Figure 11 and described further below.

The VMM 107 schedules the virtual machines, VM A 103 and VM B 105, for execution on the bare machine 101 in a fashion that allows the VMs to share the computing resources of the bare machine 101. Each VM allocate its shares of the computing resources to its guest operating system (OS) and to any user-level applications running in that particular VM, such as guest OS1 109 and applications 113 within VM A 103, and guest OS2 111 and applications 115 for VM B 105. The actual allocation of the computing resources by the VMM 107 depends, at least in part, on the particular embodiment of the virtual machine system 100 and the applications 113, 115 being run by the guest OSs 109, 111 within the VMs 103, 105. In particular, when one or more of the applications 113, 115 are real-time applications, the VMM 107 must allocate the computing resources to provide adequate scheduling Quality of Service (QoS) to the real-time application(s), i.e., the VMs are scheduled for execution with sufficient frequency and for sufficient durations that the deadlines for their real-time applications can be met.

Real-time systems theory and practice teaches that a real-time application thread (or other schedulable entity such as a process, guest operating system, virtual machine, etc.) can be guaranteed sufficient scheduling QoS by reserving a certain amount of

processor time for the real-time application, typically expressed in terms of a percentage of the processor (X) and a period of (cyclic) execution (Y). In other words, the scheduling requirements of a real-time application can be abstracted as X microseconds of execution time every Y microseconds of wall clock time. For example, to provide adequate scheduling QoS to a real-time application 113, VM A 103 might need to receive 2 microseconds of execution time on the processor of the bare machine 101 every 10 microseconds of wall clock time. Once VM A 103 has received its 2 microsecond during any 10 microsecond period, the VMM 107 saves the state of VM A 103 and switches in the state of VM B 105 for execution.

However, the actual computing resource requirements X and Y of the VMs 103, 105 are not directly accessible by the VMM 107 and thus must be inferred from events within the VMs that are visible to the VMM. Assuming that real-time processing in a guest OS is interrupt driven via a periodic clock interrupt (or other periodic interrupt), the VMM 107 can deduce the computer resource requirements for a VM executing a real-time guest OS or applications based on the instruction stream executed by the VM since instructions can be trapped by the VMM (illustrated by arrows from the guest OSs 109, 111 and the applications 113, 115 to the VMM 107 in Figure 1). In particular, detection of real-time process and/or thread switches within the VMs 103, 105 enables the VMM 170 to monitor scheduling decisions by the guest OSs 109, 111 and to determine if the current computing resource requirements allow the VMs to provide adequate scheduling QoS to their real-time applications.

Within a VM, when the active process or thread completes its work, or blocks before completing its work, its state is saved by the guest OS and the execution is switched to another process or thread. OSs that use static priority schedulers, such as Windows NT,

typically assign higher priorities to processes and threads belonging to real-time applications than to processes and threads belonging to non real-time applications, so that any ready-to-run real-time threads or processes will complete or block before any non-real-time threads or processes begin executing. Typically, when a VM resumes execution from being switched out by the VMM for a period of time, a number of interrupts will be pending, including clock interrupts, and the OS will tend to make rescheduling decisions immediately after being switched in as threads and processes that were blocked waiting on those interrupts are unblocked. Thus, the VMM can infer the processes and threads belonging to real-time applications by monitoring which processes and thread are switched in first within each VM over time. Alternatively, for OSs such as Windows NT that do not implement a priority inheritance protocol the VMM can utilize a "helper" thread or process at a low real-time priority that is set up to execute in each VM to establish a "fence." If, following an interrupt, one or more threads and/or processes are switched in before the helper thread or process is switched in , and there has been no intervening interrupt, then the VMM can be assured that those threads and/or processes in the VM are real-time.

To deduce the periodic frequency (Y) for VM A 103, for example, let Y_A be the minimum Y for all real-time applications being executed by the guest OS1 109. Given that all processing in VM A 103 is driven by a clock interrupt having a rate T_A, then $Y_A >= T_A$. The calculation of Y in a system driven by some other periodic interrupt source is analogous. By incorporating a component that detects interrupt frequency into the VMM 107, the VMM 107 can track interrupt frequency for each VM and can thus deduce a Y for each VM. Cases where a combination of periodic interrupt sources are used can be resolved by choosing the smaller period (or more generally, the GCD or greatest common

divisor), or other arbitration scheme that results in a period that closely matches the rate.

In one embodiment, the determination of how much execution time (X) to allot to each VM uses a feedback loop in which the process/thread switches for real-time applications within the VM serve as the feedback. Any of a number of well-known feedback loops can be used to close the loop in a self-dampening way and such techniques have been successfully applied to real-time software. The VMM 107 deduces X for each VM by initially assigning the VM a provisional "MHz rating," setting X to match the provisional rating, and monitoring each VM to determine if it is getting enough execution time to provide adequate scheduling QoS to its real-time processes/threads. If the active real-time process/thread is switched out before the interrupt at X occurs, then the real-time process/thread has met its deadline(s) without needing all the execution time currently assigned to the VM and the real-time application has received adequate scheduling QoS from the guest OS and from the VM. Conversely, if the real-time process/thread is still executing when it is interrupted at X, then the real-time process/thread needs more execution time. By monitoring the frequency with which real-time processes/threads within a VM receive adequate scheduling QoS, the VMM 107 can make necessary adjustments to the value of X for the VM. The VMM 107 can also detect the frequency with which a guest OS enters an idle loop, i.e., has no useful work to do, by detecting halt (HLT) instructions. Because the VMM 107 detects process and thread switches at the operating system and application levels, it can calculate its scheduling for a VM at granularities beneath the whole VM.

Figures 2A-C illustrate methods performed by the processor of the bare machine 101 to track schedulable entities in the systems 100, 150. The processor is assumed to manage the state of OS processes and threads, VMs controlled by a VMM, and

processes/threads within VMs, using one or more state registers, which will typically be protected and thus only accessible by privileged software. The processor is further assumed to have the ability to detect attempts by software executing at any privilege level to modify the protected state registers. The term "protected state register" is used generically to refer to any data structure that changes upon a process or thread switch or upon a transition from one VM to another. Specific values in the protected state registers are associated with a particular process, thread or VM. Different processors will have different data structures that can serve as the protected state registers and various embodiments are described below.

In an exemplary embodiment illustrated in Figure 3, a protected state register 363 contains an identifier for the currently executing OS process or thread, or VM controlled by a VMM, while one or more state match register(s) 357 contain an identifier for a process, thread or VM to be tracked. The state match register 357 enables specific operations to occur when the value in the state register matches (or fails to match) the value in the state match register as explained further below.

Figure 2A illustrates a method in which operations 250 performed by the processor of the bare machine 101 track process and thread switches within OS 157 in Figure 1A and transitions between VMs 103, 105 in Figure 1B. When software executing on the bare hardware attempts to modify the protected state register, e.g., register 363, at block 251, the processor determines if state tracking is active (block 253). If it is not, the processor modifies the protected state register at block 255. On the other hand, if state tracking is active, the processor checks whether one (or more) state match registers, e.g., register 357, match the current (i.e., old) value of the protected state register. For each match, the processor disables state tracking because the currently tracked process, thread or VM is

being switched out (block 261). The processor modifies the protected state register at block 259, which effects, respectively, a process, thread or VM switch. The processor checks whether one (or more) state match registers match the current (i.e., new) value of the protected state register. For each match, the processor enables state tracking for the new process, thread or VM at block 267. The statistics obtained by the state tracking can be input into various performance tools used to optimize the performance of the hardware.

One embodiment of the processes represented by blocks 261 and 267 in Figure 2A is now explained in conjunction with Figure 3. The processor of the bare machine 101 has previously configured a performance event configuration register 355 to turn on counting of performance events by configuring functional unit block 361 to signal performance events (e.g., instructions retired) to performance event count register 351. By only asserting the signal from the state match register 357 to an And gate 353 when there is a match between the state match register 357 and the protected state register 363, the performance event count register 351 will only count events occurring when the matched process, thread or VM is executing, thus tracking the schedulable entity. At block 261, the signal from the state match register 357 to And gate 353 is de-asserted to disable the counting of performance events by the count register 351. Similarly, at block 267, the signal from the state match register 357 is asserted to enable the counting of performance events by the count register 351. For clarity, a "state tracking on" register is not shown in Figure 3 and state tracking is thus implicitly assumed to be always on in such an embodiment.

In an alternate embodiment not shown, the effect of a state match could be the reverse of that illustrated in Figure 3, so that a match inhibited counting of performance events and a non-match enabled them by swapping the actions performed at steps 261 and

267 of Figure 2A.

Figures 2B and 2C illustrate a method in which operations 200 performed by the processor of the bare machine 101 work in conjunction with operations 210 performed by the VMM 107 to track process and thread switches within VMs 103, 105 in Figure 1B. The guest OSs 109, 111 are assumed to be executing on their VMs without privilege to access the relevant protected state register(s) (e.g., not in ring 0 or in a guest processor mode designed for VMMs). As disclosed in the related application serial number 09/752,587, one embodiment of a guest processor mode allows a guest OS to run at its intended privilege level, i.e., ring 0, for most operations but transfers control over operations that may result in access of certain privileged hardware resources, such as the protected state registers, to the VMM. A look-up table correlates the values in the state register with a process/thread identifier specific to the VMM and a status table records the VMM identifier of the currently active process/thread, enabling the VMM to track execution of individual guest OS processes and threads on an ongoing basis. An exemplary embodiment of a look-up table 400 is illustrated in Figure 4A and an exemplary embodiment of a status table 410 before and after a process/thread switch is illustrated in Figure 4B. Figures 4A-B are discussed in conjunction with Figure 2C.

Beginning with Figure 2B, when software executing in a VM attempts to modify a protected state register (block 201), the processor determines if the software is privileged (block 203). If so, such as when the attempt is by the VMM 107 itself, the processor modifies the protected state register at block 205. On the other hand, if the software is not privileged, such as the guest OS 109, 111 or the user-level applications 113, 115, at block 207 the processor traps the attempt to privileged software, the VMM, which causes control to be passed to the VMM 107 at block 209 ("virtualization trap").

Turning now to Figure 2C, when the VMM receives control, it uses the contents of the protected state register to determine the VMM identifier for the current process/thread through the look-up table 400 and marks the current process/thread VMM identifier as not active in the status table 410 (block 211). Referring to Figure 4A, each entry 401 in the look-up table 400 contains a state register field 403 and its associated VMM identifier field 405 so the values in the state register identify the appropriate entry 401 and the corresponding VMM identifier 405, e.g., 12320000. Referring to Figure 4B, each entry 411 in the status table 410 contains a VMM identifier 413 and a status indicator 415. In the entry 411 associated with the current process/thread, e.g., VMM identifier 12320000, the status indicator 415 is set to a value indicating that the process/thread is active. When the processing at block 211 is completed, the status indicator 415 associated with the VMM identifier 1232000 is reset, indicating that the process/thread is no longer active. It will be appreciated that a single bit can be used as the status indicator 415. In one embodiment, the look-up table 400 and the status table 410 are stored in memory; in an alternate embodiment, they are stored on a mass storage device, such as a hard disk.

The VMM modifies the protected state register on behalf of the unprivileged software (block 213), which signals a process/thread switch within the currently executing VM. The VMM determines the VMM identifier, e.g., 12330000, for the new process/thread through the look-up table 400 using the contents of the state register after it has been modified (block 215). The VMM also marks the corresponding VMM identifier as active in the status table 410 (block 217). Thus, for example, the processing at block 217 marks the process/thread VMM identifier 1233000 as active in the status table 410 in Figure 4B. The VMM resumes execution of the virtual machine in which the new process/thread will execute at block 219.

As stated above, the data structures that can serve as the protected state registers vary from processor to processor. One embodiment of the invention can be used with processors that provide a virtual addressing space to executing processes. One or more "address space" registers containing the values for the current address space must be updated on an operating system process switch. For example, when executing on an Intel IA32 processor, the VMM 107 can detect operating system process switches by monitoring changes to a control register (CR3) that contains the base address of the current page table. Similarly, the VMM 107 executing on an Intel IA64 processor can detect operating system process switches by monitoring changes to region registers that map virtual address regions into a global address space. Processors with hardware-managed address translation look aside buffers (TLBs) typically have one or more registers which are generally protected from modification by unprivileged software. Thus, VMM 107 can track changes to these registers to effectively monitor operating system process switches. Processors with software-managed TLBs also provide hardware to assist with address translation. For instance, the MIPS R10000 RISC processor associates an address space identifier (ASID) with each TLB entry to avoid costly TLB flushes upon context switches. A special register containing the ASID of the current process must be updated upon every context switch so the VMM 107 could track changes to this register to detect operating system process switches on a R10000 or similar processor.

On any processor that supports multi-threaded processes, each thread is associated with an instruction stack and a stack pointer that points to the current top of the stack. Typically, the guest OS 109, 111 will store the stack pointer for a non-executing thread in a data structure in memory (often called a thread control block). By monitoring loads or stores of the stack pointers from or to memory, a processor could detect thread switches.

However, current processors do not protect the stack pointers from access by unprivileged software, and thus do not fault when load/store instructions on the stack pointers are issued by unprivileged software. By modifying the processor to protect the stack pointers in such a fashion, the stack pointers would serve as the protected state registers and the VMM 107 could detect memory loads/stores of the stack pointers by the guest OSs 109, 111, thereby monitoring thread switches in the VMs 103, 105. Although such changes could be accomplished without impacting the usability of the stack, they would likely make implementation of light weight user-level thread packages problematical. Moreover, microarchitecture design considerations and microcode implementation issues make it difficult to modify many standard processors, such as the IA32 and IA64, to fault in this fashion.

In some processors, such as the IA32, the current instruction stack is stored in a segment and identified by a segment selection in a stack segment register. Because certain operating systems, such as the Microsoft Windows 9x family and some optimized real-time operating systems, leverage the processor's segmentation architecture for low cost address space protection, they must modify the stack segment register when switching threads. Thus, in yet another embodiment of the invention, the processor would be modified to fault when unprivileged software attempts to load a stack segment register, enabling the VMM 107 to track thread switching in these operating systems executing on a segmented architecture processor.

Other processors, such as the IA64, incorporate instruction level parallelism that uses speculation techniques to determine the next instructions and data most likely to be required by the processor. The processor uses a data structure to hold the data speculative state of the processor, referred to hereinafter as the Advanced Load Address Table

(ALAT). Entries in the ALAT are invalidated by events that alter the state of the processor and such events are relatively well-coordinated with thread switches. Therefore, the ALAT could serve as the protected state registers if the processor is modified to fault on ALAT invalidations caused by unprivileged software. This would enable the VMM 107 to track thread context switches performed by both guest OSs 109, 111 and user-level applications 113, 115. It will be appreciated that although ALAT is the name of the data structure within the IA64 processor, the invention is not limited to use only with that particular data structure.

One embodiment of VMM 107 that bases its scheduling of virtual machines, at least in part, on resource requirement values derived from process/thread switches is illustrated in Figure 5. The VMM 107 includes an idle detector 501, a proportional integral derivative (PID) controller 503, and a scheduler 505. Any of these components may be implemented in software, hardware or firmware, or in any combination thereof. In one embodiment, the VMM 107 operates according to a method 600 shown in Figure 6.

When the VMM 107 receives a virtualization trap for VM A 103 or VM B 105 from the processor as previously described in conjunction with Figure 2B, it notifies the idle detector 501. The VMM 107 may also receive halt instructions (HLT) that indicate the guest OS executing within a VM has no useful work to do and this information is also passed to the idle detector 501. The VMM 107 performs operations at blocks 603, 605, 607 and 609 as previously described in conjunction with Figure 2C. At block 611, the VMM 107 runs the PID controller 503 and the scheduler 505. PID controllers are typically employed in feedback loops to iteratively derive a second value from a measured first value and are well-known by those skilled in the art. While the invention is not limited in scope to the use of any particular configuration of a PID controller, in the

particular embodiment shown in Figure 5, the idle detector 501 passes the measured value to the PID controller 503 as a binary, e.g., did a switch occur before end of time slice or did all real-time processes/threads run on a VM, or a scalar, e.g., a number of switches before a HLT on a VM. The PID controller 503 determines the VM resource requirement value X for the VM as described previously. The resource requirement value X is used by the scheduler 505 to determine the schedule for the VM. At block 613, the VMM 107 schedules the execution of the appropriate VM in accordance with the output of the scheduler 505.

Thus, the current invention enables a VMM to monitor scheduling decisions by guest operating systems. Based on the characteristics of various guest operating systems, the VMM can infer scheduling QoS at the level of the operating system process or even individual threads. For a general purpose operating system such as Microsoft Windows NT and Windows 2000, this could allow the VMM to track execution of threads executing with real-time priority by inferring processes in the Win32 real-time priority class and ensure that threads in those processes received adequate scheduling quality of service. Thus, detecting process/thread switches can enable a VMM to better schedule a system in which general purpose OSs are executing both real-time and non-real-time applications within VMs.

A recursive virtualization environment is illustrated in Figure 7, in which a child VMM 701 operates at a non-privileged level (i.e., not at ring 0) to schedule virtual machines A1 703 and A2 705 within the virtual machine A 103. Many current processors do not include hardware support for recursive virtualization, thus all state related to recursive virtualization must be maintained by software instead of through hardware registers. Additionally, maintaining the state of multiple levels of virtualization in

software and transitioning among those states becomes increasingly difficult as more levels of recursion are added. Therefore, some form of hardware support for virtualization is required to successfully achieve recursive virtualization.

In one embodiment, no special hardware support is provided for virtualization so the VMM 107 executes with full privilege (e.g., in ring 0) and both the guest OS 111 and guest applications 115 execute without privilege (e.g., in ring 3). In one embodiment on such hardware the VMs 103 and 105 have separate address spaces so that guest OS 109 and guest OS 111 are protected from one another. On such hardware the child VMM 701 would thus execute without privilege (e.g., in ring 3), as would guest OS 3 707, guest OS 4 709 and applications 711 and 713. Since the parent VMM 107 executes with full privilege it can switch processes on behalf of guest OS 1 111 executing within VM B 105 and it can switch execution to VM A 103 without having those instructions trapped by the processor. However, the child VMM 701 cannot switch processes on behalf of guest OS 3 707 executing within VM A1 703 nor can it switch execution from VM A1 703 to VM A2 705 because the child VMM 701 is in fact executing without privilege. As noted earlier, in one embodiment the parent VMM 107 transitions from VM A 103 to VM B 105 by effecting a process switch, but as noted earlier the child VMM 701 executes at ring 3 and so an attempt by it to switch from VM A1 703 to VM A2 705 by switching processes will trap to the parent VMM 107. As a result all attempts by the child VMM 701 to switch VMs trap to the parent VMM 701, which is thus enabled to track VM switches by the child VMM 701.

In an alternate embodiment, the child VMM 701 executes in a non-privileged virtualization mode on a processor that includes a hardware VM-state register and a VM-run instruction. The VM-state register is loaded with the state information for a virtual

machine and the VM-run instruction causes the processor to begin execution of that state, analogous to a state register for a process/thread and the transition to a privilege level in which the execution of a process/thread is invoked. Thus, the processor could trap either an instruction to load the VM-state register or the VM-run instruction to track VM switches by a child VMM 701. However, many of the executions of the VM-run instruction will not be as a result of a change in VM but will be due to attempts by a guest OS to execute a privileged instruction, page faults, etc. By trapping, instead, on loads of the VM-state register, the processor could ensure that the parent VMM 107 only receives virtualization traps from the child VMM 701 in response to actual VM switches by the child VMM 701.

As shown in Figures 8A and 8B, at block 801 the processor 101 could trap either the issuance of an instruction to change the contents of the VM-state register or the issuance of the VMM-to-VM mode transition instruction. If the instruction was issued by software not privileged for the instruction (i.e., a child VMM 701), the processor 101 would trap to privileged software, the parent VMM 107, at block 807 and jump to the parent VMM 107 at block 809. The parent VMM 107 would maintain machine look-up and status tables for the various VMs as shown in Figures 9A-B, which are analogous to the corresponding data structures described above for processes and threads. The parent VMM 107 would mark the current VM identifier as not active in the machine status table 910 (block 811), perform the appropriate instruction on behalf of the child VMM 701 (block 813), determine the identifier for the new VM using the machine look-up table 900 and mark it as active in the machine status table 910 (blocks 815 and 817), and resume execution of the child VMM 701 at block 819.

The VMM 107 of Figure 5 can schedule VMs for child VMM 701 in a recursive

virtualization environment using a method 1000 shown in Figure 10. At block 1001, the idle detector 501 is notified when the VMM 107 receives virtualization traps from the processor in Figure 8A. The operations performed by blocks 1003, 1005, 1007 and 1009 are as described above in conjunction with blocks 811, 813, 815 and 817 in Figure 8B. At block 1011 the PID controller 503 and scheduler 505 are run to process the information from the idle detector 501 and the execution of the child VMM 701 is resumed at block 1013. The child VMM 701 subsequently schedules the appropriate VM according to the schedule calculated by the scheduler 505. Again, the parent VMM 107 would still track halt instructions as well, but the idle detector would now receive idle indications at, for example, the granularity of VMs of the child VMM 701 (i.e., VMs inside a VM).

Thus, the current invention enables a parent VMM 107 to monitor machine scheduling decisions made by a child VMM 701. The parent VMM can use this information to schedule a child VMM 701 for execution in such a way that the child VMM 701 is able to schedule its VMs as may be necessary in order that applications executing in VMs of the child VMM 701 receive adequate scheduling quality of service.

The following description of Figure 11 is intended to provide an overview of a processing system in which embodiments of the invention can be implemented, but is not intended to limit the applicable environments. Figure 11 illustrates one example of a conventional computer system containing a processing unit 1151 that incorporates supports the execution of a virtual memory monitor of the present invention through hardware, firmware, or software. Memory 1159 is coupled to the processor 1155 by a bus 1157. Memory 1159 can be dynamic random access memory (DRAM) and can also include static RAM (SRAM). The bus 1157 couples the processor 1155 to the memory 1159 and also to non-volatile storage 1165 and to display controller 1161 and to the

input/output (I/O) controller 1167. The display controller 1161 controls in the conventional manner a display on a display device 1163 which can be a cathode ray tube (CRT) or liquid crystal display. The input/output devices 1169 can include a keyboard, disk drives, printers, a scanner, and other input and output devices, including a mouse or other pointing device. The display controller 1161 and the I/O controller 1167 can be implemented with conventional well known technology. A digital image input device 1171 can be a digital camera which is coupled to an I/O controller 1167 in order to allow images from the digital camera to be input into the computer system 1151. The non-volatile storage 1165 is often a magnetic hard disk, an optical disk, or another form of storage for large amounts of data. Some of this data is often written, by a direct memory access process, into memory 1159 during execution of software in the computer system 1151. One of skill in the art will immediately recognize that the term "computer-readable medium" includes any type of storage device that is accessible by the processor 1155 and also encompasses a carrier wave that encodes a data signal.

Techniques for detecting in hardware transitions among software processes or threads or among virtual machines have been described. In addition, techniques for detecting in a virtual machine monitor transitions among software processes or threads or virtual machines belonging to a child virtual machine monitor have been described. Embodiments of hardware performance monitoring counters that utilize these techniques to distinguish events that occur in one process, thread or virtual machine from those that occur in another have also been described. Finally, embodiments of a virtual memory monitor that utilize those techniques to provide adequate scheduling quality of service to a real-time applications executing within the virtual machines have been described. Although specific embodiments have been illustrated and described herein, it will be

appreciated by those of ordinary skill in the art that any arrangement which is calculated to achieve the same purpose may be substituted for the specific embodiments shown. This application is intended to cover any adaptations or variations of the present invention. For example, those of ordinary skill within the art will appreciate that the embodiments of the invention have been described above as switching between two schedulable entities for ease in explanation and the invention is not limited to virtual machine environments in which only two schedulable entities are executing. Therefore, it is manifestly intended that this invention be limited only by the following claims and equivalents thereof.